13th ICCRTS: C2 for Complex Endeavors

"High Productivity Computing Systems for Command and Control"

Topic: C2 Architectures

Authors: Scott Spetka, SUNYIT and ITT Corp

Authors: Christopher Flynn, Air Force Research Laboratory

Point of Contact: Scott Spetka

Name of Organization: SUNYIT and ITT Corp.

Complete Address: Computer Science Department, SUNYIT, Route 12 North, Utica, NY, 13504-3050

Telephone: 315-792-7354

E-mail Address: scott@cs.sunyit.edu

Report Documentation Page		Form Approved OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **JUN 2008**	2. REPORT TYPE	3. DATES COVERED **00-00-2008 to 00-00-2008**
4. TITLE AND SUBTITLE **High Productivity Computing Systems for Command and Control**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **SUNYIT,Computer Science Department,Route 12 North,Utica,NY,13504-3050**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
13th International Command and Control Research and Technology Symposia (ICCRTS 2008), 17-19 Jun 2008, Seattle, WA

14. ABSTRACT

The most significant issue underlying all future command and control (C2) architectures is the ability to develop software that can harness the next generation of processors. Multicore processors, scaling into thousands of processors per chip will soon be prevalent in all C2 systems. The success of C2 systems will depend on our ability to adapt to the new processor technology. Existing C2 systems that implement scientific codes for image processing and many other applications have been a dominant user of high performance computers (HPCs) for several decades. However, increasingly diverse C2 applications are now also being adapted to HPCs, due to dropping prices and increased availability. The goal of the DARPA High Productivity Computing Systems (HPCS) program is to develop high performance computers that are substantially easier to program, thereby reducing software development cost and time to solution. We employ a publication/subscription information management (PSIM) system in a case study to compare new HPCS approaches to parallel code implementation with existing techniques. The PSIM system requires intensive CPU cycles and communications bandwidth, for brokering XML information objects between publishers and corresponding subscribers. The study compares two new HPCS languages, Chapel (Cray) and X10 (IBM), with the Message Passing Interface (MPI) standard.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**24**	

High Productivity Computing Systems for Command and Control

Abstract

The most significant issue underlying all future command and control (C2) architectures is the ability to develop software that can harness the next generation of processors. Multicore processors, scaling into thousands of processors per chip will soon be prevalent in all C2 systems. The success of C2 systems will depend on our ability to adapt to the new processor technology. Existing C2 systems that implement scientific codes for image processing and many other applications have been a dominant user of high performance computers (HPCs) for several decades. However, increasingly diverse C2 applications are now also being adapted to HPCs, due to dropping prices and increased availability.

The goal of the DARPA High Productivity Computing Systems (HPCS) program is to develop high performance computers that are substantially easier to program, thereby reducing software development cost and time to solution. We employ a publication/subscription information management (PSIM) system in a case study to compare new HPCS approaches to parallel code implementation with existing techniques. The PSIM system requires intensive CPU cycles and communications bandwidth, for brokering XML information objects between publishers and corresponding subscribers. The study compares two new HPCS languages, Chapel (Cray) and X10 (IBM), with the Message Passing Interface (MPI) standard.

Keywords: pub/sub, chapel, X10, MPI, HPCS

Introduction

This paper describes an experimental implementation of a prototype publication/subscription (pub/sub) information management system (PSIM) that is being used as an example to compare implementation approaches for the two new high productivity computing systems (HPCS) languages, Chapel and X10, with a legacy Message Passing Interface (MPI) approach to parallel computing. The HPCS languages and their implementations are still evolving and improving at a rapid pace. Some changes to the features discussed in this paper can be expected. Pub/sub technologies have been shown to be important to net-centric warfare, for command and control [1], making a pub/sub system an especially relevant choice for a language comparison in the context of command and control. The pub/sub model system being implemented is shown in the diagram below (figure 1).

The first section describes the model that will be implemented in each of the three languages. The next section describes the two main requirements for a parallel implementation of pub/sub that we will study. Then, in the next three sections, we focus on the HPCS languages and MPI, comparing them and contrasting them, in the context of our pub/sub model, and presenting some of the highlights of our implementation for each approach. In conclusion, we make observations from our experience working with the

three languages and discuss the significance of the work for command and control applications.

Figure 1 - Pub/Sub Model
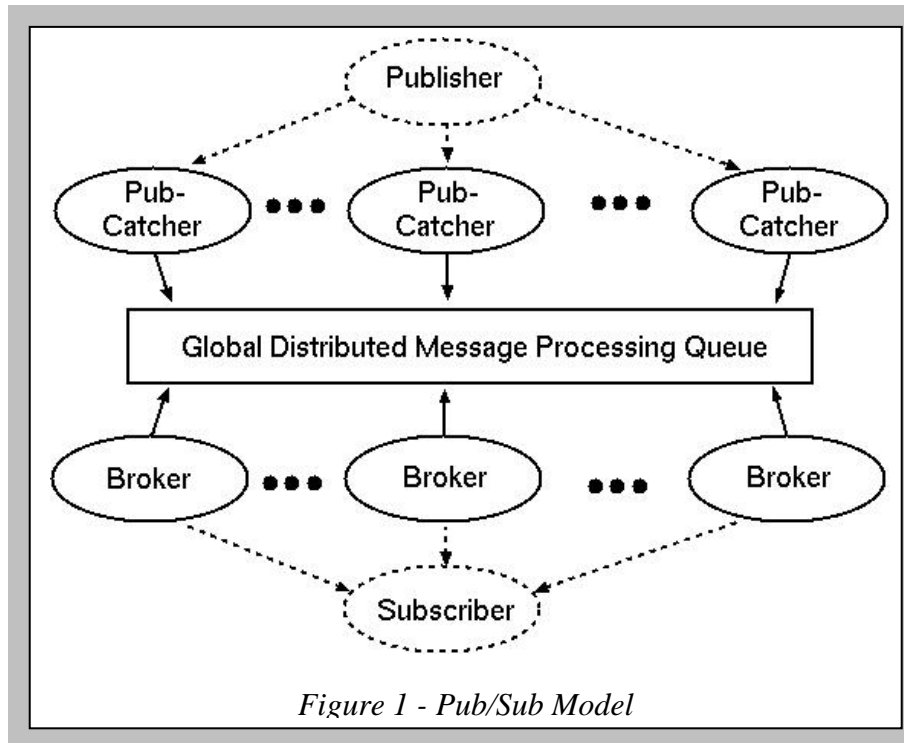
The Pub/Sub Model

The main characteristic of the pub/sub system, shown in Figure 1 above, is the decoupling of publishers from subscribers. Publishers do not need to know who is subscribing and subscribers do not need to know who is publishing [2]. In place of static network connections, brokers work to match publications with their subscribers. Pub/sub systems are ideally suited to C2 systems where units may enter a battlefield during the course of an ongoing battle and may also be abruptly removed from a battlefield. Such a system would be extremely difficult to build upon a point-to-point connection-oriented model.

Subscribers use XPATH expressions to specify the contents of XML documents [3] that they are interested in and that may be published. Brokers must evaluate the XPATH predicates for each document received, to determine which subscribers should receive the document and then disseminate the document to the interested subscribers. The pub/sub model is very robust and may be designed to accommodate system failures in parallel and distributed systems. The notion of routing data, based on content provides a lot of flexibility for implementing parallel support for pub/sub systems. High performance computers are used to implement brokering functions for scalable systems, to prevent performance degradation when brokering loads are high. This makes brokering documents an excellent target for experimentation with new languages for high performance computing.

In the pub/sub model, publishers send XML documents (publications) into the system via "pubcatchers" (see Figure 1). Pubcatchers insert the documents into a globally visible queue, where brokers can access them. Brokers have access to subscriptions, which are expressed as XPATH predicates. The predicates are used to determine which subscribers should receive each document that is brokered. In the diagram, the dotted lines for publishers and subscribers indicate that there may be many publishers and many subscribers. Each brokered document is delivered to a subset of subscribers.

Comparing Implementation Approaches

In figure 1, a global transparent message-processing queue forms the heart of the system. The main issue confronting a parallel system designer, for pub/sub systems is to provide efficient access to this distributed data structures, in this case the global message queue. The two main issues that we consider, in comparing the three approaches to parallel implementation of pub/sub, are data distribution and synchronization. They are the key aspects of achieving parallelism in most implementations.

In Chapel and X10, data distributions are defined separate from variable declarations, allowing changes to distributions to be made without changing the code that accesses the data. All processes see a global partitioned address space (GPAS) that is a single address space distributed across processors. In MPI there is a cooperative view of data that is partitioned across processors, each processor must know exactly where each element of a distributed array is located.

Synchronization is the other major feature of any parallel programming language that has the most impact on programmer productivity. Easy access to a globally visible array makes synchronization among processes more critical. It makes it more likely that remote and local accesses will conflict, simply because writing code that accesses remote objects is transparent and therefore easier. In Chapel and X10, modification of data can be accomplished transparently, simply by specifying a loop that accesses the program data elements, the pub/sub message queue in our example. In MPI, modification of data on remote systems must be accomplished via explicit messages. In most cases, code has to be written for both senders and receivers for data exchange among MPI processes.

In addition to comparing language features, with respect to support for synchronization and data distribution, other aspects of the programming environments supported for each approach should be considered, especially when evaluating new technologies, like Chapel and X10. Some HPCS language features were either not available or were not fully implemented in the early releases used in this study. Figure 2 also compares Chapel, X10 and MPI in terms of documentation, object-oriented paradigm, running the codes and sharing data among processes. The next three sections focus on each of the three languages.

Issue	Chapel	X10	MPI
Documentation/Help	Ongoing email group. Some examples.	Newsgroups. A lot of examples.	Mature Documentation
Data Distribution	Define data distributions separate from variable declarations	Define data distributions separate from variable declarations	Collections are built from local component data
Object-Oriented	Inheriting from multiple base classes, with restrictions (true multiple inheritance is not yet supported)	Interfaces (Java approach to multiple inheritance)	Used Mpich2.0 not OO (OOMPI or MPI C++ also possible)
Synchronization	"Sync" variables and "atomic" statement (atomic not yet supported)	"async" and "finish" operations on remote data.	"barriers" and "communicators" for brokers and publishers.
Running the Code	Type the compiled executable name. Stdin/stdout supported	Use "testScript", find output in log and error files.	Use Mpich2 scripts to start mpd and mpiexec codes.
Sharing Data	Transparent Global Partitioned Address Space (GPAS)	Transparent Global Partitioned Address Space (GPAS)	Single reader "broadcasts" data to workers nodes. Workers share data by "send", "receive" and "broadcast".

Figure 2 – Language Comparison

Chapel

The Chapel language [4][5] is being developed by Cray Inc. [6] under the DARPA High Productivity Computing Systems (HPCS) program. The Chapel distribution includes a set of example programs. The Chapel producer/consumer example program was used as a starting point for the Chapel pub/sub model implementation. We experimented with Chapel language features that support methods for partitioning and distributing data.

Although true distribution across remote "locales" was not supported in the early release of the language that we used in this study, we were able to define distributions that were mapped to local data storage. We also focused on synchronization, using Chapel sync variables [7]. Sync variables can be used to control concurrency and avoid conflicting updates on shared variables. Reading a sync variable essentially block all other readers until a write takes place, which releases the lock that is granted exclusively to a single reader.

In the initial implementation, we are implementing a queue of publications. Pubcatchers insert the publications into sequential locations in the queue. The queue is cyclic distributed so that incoming publications will be distributed across systems to evenly distribute the brokering load while allowing both brokers and pubcatchers to operate primarily on their local segment of the queue. In this implementation, we wanted to use Chapel sync variables to enforce a strict brokering order. We are using sync variables to control an alternating pattern of execution, with the pubcatchers and brokers synchronizing their accesses independently based on independent sync variables.

The Chapel data distribution features [8] have a major impact on developer productivity. Using the Chapel built-in cyclic distribution made this approach easy to implement and saved a considerable programming effort that would have been needed for an MPI implementation. We are currently building an MPI implementation, but implementing a distributed queue in MPI will require global synchronization among processors using only "barriers". MPI barriers require all cooperating processes to arrive at the barrier before any of them can continue executing, thereby assuring that the data protected by the barrier has been updated by all processes before execution continues. Also, in Chapel, access to the local array elements is transparent to the user and efficiently compiled. In MPI, expensive runtime evaluation of user-defined data types may be required to move date for communication.

The initial Chapel implementation of the pub/sub model has two brokers and two pubcatchers. Pubcatchers use a shared counter variable to access the publication queue. There are also two brokers that share a counter for access to the publication queue. Synchronization worked well and was also easy to use. We implemented critical sections by using sync variables. This approach can be easily extended to handle more than two publishers and two brokers. Atomic blocks, another Chapel language feature for synchronization could also be used, but it is not yet implemented.

Initiating parallel computations in Chapel is made easy through the use of "cobegin". A cobegin block automatically launches one process each for each function specified in a code block, designated by: "cobegin { [list of function calls] }". For the pub/sub model, we used a cobegin inside a loop to start the desired number of publishers and subscribers for each run. Cobegin can launch arbitrary statements, including compound/block statements. The cobegin language feature of Chapel, along with its flexible data distribution mechanism provide independence from the requirement to use a fixed number of processors throught an execution, even when the processing requirements may increase or decrease. Eight of the Chapel language features, illustrated by our implementation, and Chapel specific language issues are described in figure 3.

1) Offers a variety of mechanisms that can assure cooperating processes are synchronized.

2) *Sync* variables were used to synchronize producers and consumers. *Sync* and *single* variables are easy to use and an interesting feature of the Chapel language.

3) Could not use Chapel "atomic" sections because they are not yet implemented.

4) Time related functions, like *sleep*, are supported through a Time module. External function call mechanism also supported.

5) Added synchronization for multiple brokers.

6) Easy to extend synchronization for more complex pub/sub model.

7) Easy to extend pub/sub example code to experiment with other built-in data distributions.

Figure 3 – Chapel Language Experience

X10

The X10 programming language [9][10] has many concepts that are similar to Chapel. It supports a global partitioned address space (GPAS), provides mechanisms for creating data distributions that are independent of access, and provides synchronization mechanisms that aim to simplify the task of developers and make them more productive. It was easy to create new distributions at all "places" or at a subset of processes involved in the computation. Like Chapel, definition of places for data distribution and the distribution algorithm, like block cyclic distribution, are orthogonal to the code implemented to access the distributed data, allowing flexibility to change distributions without rewriting the processing algorithm. X10 makes it is easy to define new distributions by using inheritance to extend existing distributions.

For the pub/sub model, we created a distribution, where the publication processing array is block distributed across all places (default is 4 places). Then we created two brokers and two pubcatchers. Pubcatchers listen for publications from publishers, currently simulated by reading them from a file, and insert them into the global publication processing queue to be brokered. Declaring the queue to be "atomic" causes X10 to synchronize pubcatcher access to the publication processing queue with other pubcatchers and brokers. We can create as many brokers as are necessary to keep up with the publication rate. Using an "atomic" queue make it easy for brokers to remove publications from the publication processing queue and to automatically synchronize with other brokers and pubcatchers. X10 also allows transparent access to remote elements in a distribution through "async" and "final" function call modifiers that can be used to control synchronized access to the remote elements

Our X10 implementation of the pub/sub model is based on an example from the X10 code distribution [9]: "Building arrays distributed across places using the union-of-

distribution approach". It illustrates, in principle, an approach that would meet the requirements of our pub/sub system. Experiments with X10 and experience using the X10 programming environment is summarized in figure 4.

1) This study was unable to get Eclipse set up yet to work with the X10 code. A new set of tools for working with X10 has been integrated into Eclipse [11], IBM's IDE, but their installation and configuration on Linux was tedious.

2) X10 has a lot of built-in functionality, including most of Java. X10 is implemented as an extension of the Java language, replacing some of its functionality. A rich set of libraries, written in Java, are available to programmers. Of course, this approach also inherited all of the disadvantages of Java for high performance computing, including garbage collection and lack of pointers.

3) We worked with array distribution code examples and then built a process distribution version. X10 has a lot of examples to work with. The runtime environment, based on their test harness was easy to work with, but put all output into a test results file. This is reasonable for checking to be sure that all test codes work correctly for each build.

4) We started with BlockDistedArray1D.x10 example code.

5) We implemented code to print the processing locations where operations take place.

6) The output from the enhanced code indicated that operations were taking place at each processor at the correct time.

Figure 4 – X10 Language Experience

MPI

The Message Passing Interface (MPI) [12] is implemented by the MPICH2 software, which is freely available on the Internet. MPI has been a standard for parallel processing for many years. Our MPI implementation of the pub/sub model distributed the publications queue across all processors. A message/cyclic distribution is used so that sequential access to the queue, for brokering messages, results in a round robin allocation of brokering of messages by processors. Brokers process messages in the queue when they are found locally in first-in/first-out order. If no new messages are in the queue, brokers at all processors are idle.

Each processor also runs a publisher that sends publications to the remote publication site that currently represents the end of the queue. Publishers must synchronize when adding publications, to avoid race conditions. Since publishers must synchronize independently of brokers, a separate "communicator" is used for publishers. Similarly, brokers have to synchronize among themselves, to maintain a common view of which entry is the next that needs to be brokered. Brokers, do not, however, communicate among themselves for other purposes.

We expect that MPI synchronization, based on communication primitives and barriers, will be less efficient than X10 and Chapel, where synchronization primitives, like sync variables in Chapel, can be efficiently compiled. The MPI model for the pub/sub implementation is shown in figure 5.

Figure 5 - MPI Fragmented Memory Model
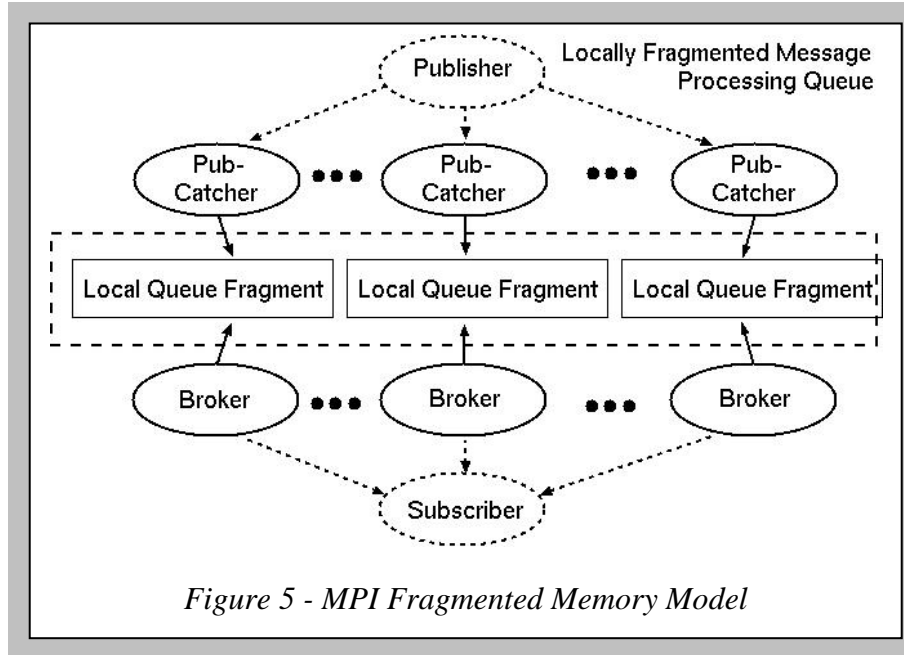
The principal problem faced by developers writing MPI code, when implementing systems that are similar to our pub/sub system is that code must be written to explicitly determine whether data is local or remote, and execute different code accordingly. To write data into a particular location in a distributed array, the MPI application must compute which processor would store the component of the array where the data is to be stored. If the data and array destination location are local, the data can be copied into the array location. Otherwise, the processor storing the data must send it to a remote process and the remote process must be waiting to receive the data. This process of coding software that implements data distribution functions in MPI is tedious and expensive. Programmer productivity is extremely low, compared to productivity for programmers using the Chapel and X10 paradigms.

When a publication arrives at a pubcatcher, we check the count to determine where the publication belongs in the message queue. If it belongs in the local fragment of the distributed message queue, we copy it to the local data array. If it is remote, we use MPI_SEND to send it to the remote processor for the component of the message queue where it is to be stored. Pubcatchers also have to listen for inputs from publishers, in addition to waiting to receive data from remote processors for storage in the local array fragment. This requires that asynchronous receives be done for MPI messages from other processors, making the coding even more difficult.

Future Work

The pub/sub model offers many alternatives to exploit its inherent parallelism. Both pubcatchers and brokers operate in parallel by processing data that is exchanged through the shared "queue", which is actually just a distributed collection that we may want to access in various ways in future experimentation. We are also interested in exploring in other directions, based on this initial design and implementation of a Chapel pub/sub information management system. Approaches that allow out-of-order processing may be most appropriate, since publications can arrive sporadically. For load balancing, pubcatchers and brokers may both need to send publications to remote locations or retrieve publications from them.

While implementing an MPI version of our pub/sub system, it became clear that writing MPI code is a lot more difficult and time-consuming than writing code in X10 or Chapel, even for a programmer who is experienced with MPI. We have implemented the Chapel pub/sub system in a variety of ways, using synchronization mechanisms and data distributions experimentally. We have also experimented with X10 and written parts of the pub/sub model in X10. We now intend to converge on a few implementations with distinct features and compare the implementation effort with an MPI-based implementation effort for a system with similar features.

Experiments performed so far have provided some confidence that using the new HPCS languages will allow developers to implement systems faster. Also, because X10 and Chapel offer a higher level at which parallelism is described, we expect that the compilers will be able to more effectively optimize the code. The data distribution features that were used in the examples above demonstrate that tuning data distribution support to improve performance can be easily done in the HPCS languages, X10 and Chapel, but would require a major effort for MPI.

Conclusion

Our experience in developing the pub/sub model indicates that synchronization in the new HPCS languages, Chapel and X10, are easy to understand and use. Compared to MPI, it is clear that programmer productivity is higher for the HPCS languages. The MPI libraries have limited features that would allow them to initiate parallel processes. The MPI "World" is defined at the beginning of execution. The HPCS languages, while initially not supporting dynamically changing process sets, are designed to offer the potential to support them in the future. We have discussed several ways in which the new HPCS languages incorporate features that are not currently available in MPI, allowing them to support improved programmer productivity.

Future C2 system architectures will depend more heavily on parallel processing and multiprocessor machines. It is fortunate that DARPA began its HPCS program several years ago, so that new HPCS languages will be ready for production deployment by the year 2010. MPI could support many scientific applications where the development costs and timeline could be tolerated as the only way to solve some problems. The expanded role of parallel computing in modern C2 systems requires rapid development and reduced

maintenance costs, both in terms of dollars and time. This paper shows that new languages have proven functionality that will help to meet that demand.

References

[1] 2006 - Spetka, S.E., Ramseyer, G.O., Tucker, S.N., Lok-Kwong, Y., Fitzgerald, D.J., Linderman, R.W., "Net-Centric Pub/Sub Information Management Design for Command and Control", Command and Control Research and Technology Symposium, San Diego, CA, June 20-22, 2006.

[2] 2002 - Combs, V., Linderman, M., "A Jini-Based Publish and Subscribe Capability", Proceedings of SPIE, Volume 4863, Java/Jini Technologies and High-Performance Pervasive Computing, June 2002.

[3] Extended Markup Language (http://www.xml.org).

[4] 2007 - Bradford L. Chamberlain, David Callahan, Hans P. Zima, "Parallel Programmability and the Chapel Language", *International Journal of High Performance Computing Applications*, August 2007, 21(3): 291-312.

[5] Chapel Home Page - http://chapel.cs.washington.edu/

[6] Cray, Inc. - http://www.cray.com

[7] Chapel Specification (version 0.750) - http://chapel.cs.washington.edu/

[8] 2007 - Roxana E. Diaconescu and Hans P. Zima, "An Approach to Data Distributions in Chapel", *International Journal of High Performance Computing Applications*, August 2007, 21(3): 313-335.

[9] The Experimental Concurrent Programming Language (X10) - http://x10.sourceforge.net

[10] The X10 Programming Language – http://www.research.ibm.com/x10

[11] Eclipse - an open development platform - http://www.eclipse.org/

[12] Message Passing Interface (MPI) - www.mcs.anl.gov/mpi/

.

High Productivity Computing Systems for Command and Control

Scott Spetka – SUNYIT and ITT Corp. – scott@cs.sunyit.edu

Christopher Flynn – Air Force Research Laboratory, Information Directorate - Christopher.Flynn@rl.af.mil

13th ICCRTS: C2 for Complex Endeavors
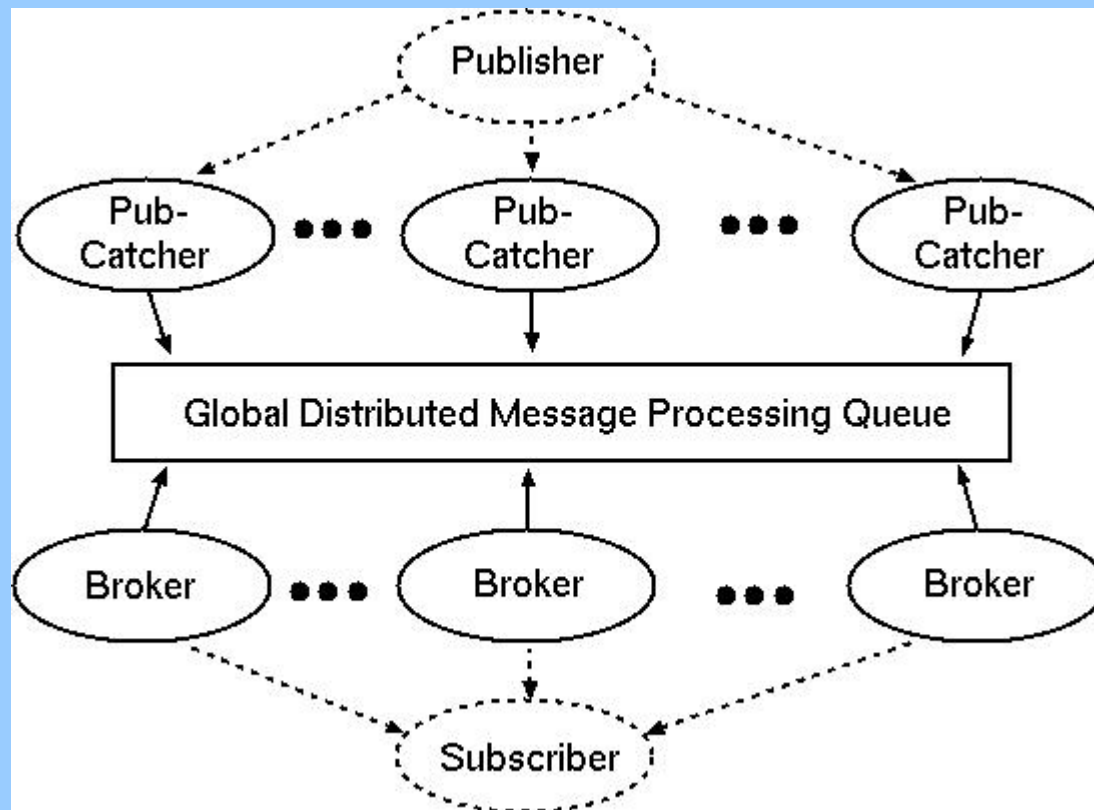
Bellevue, WA

June 17 – 19, 2008

Outline

- Pub/Sub Model

- Chapel, X10, MPI
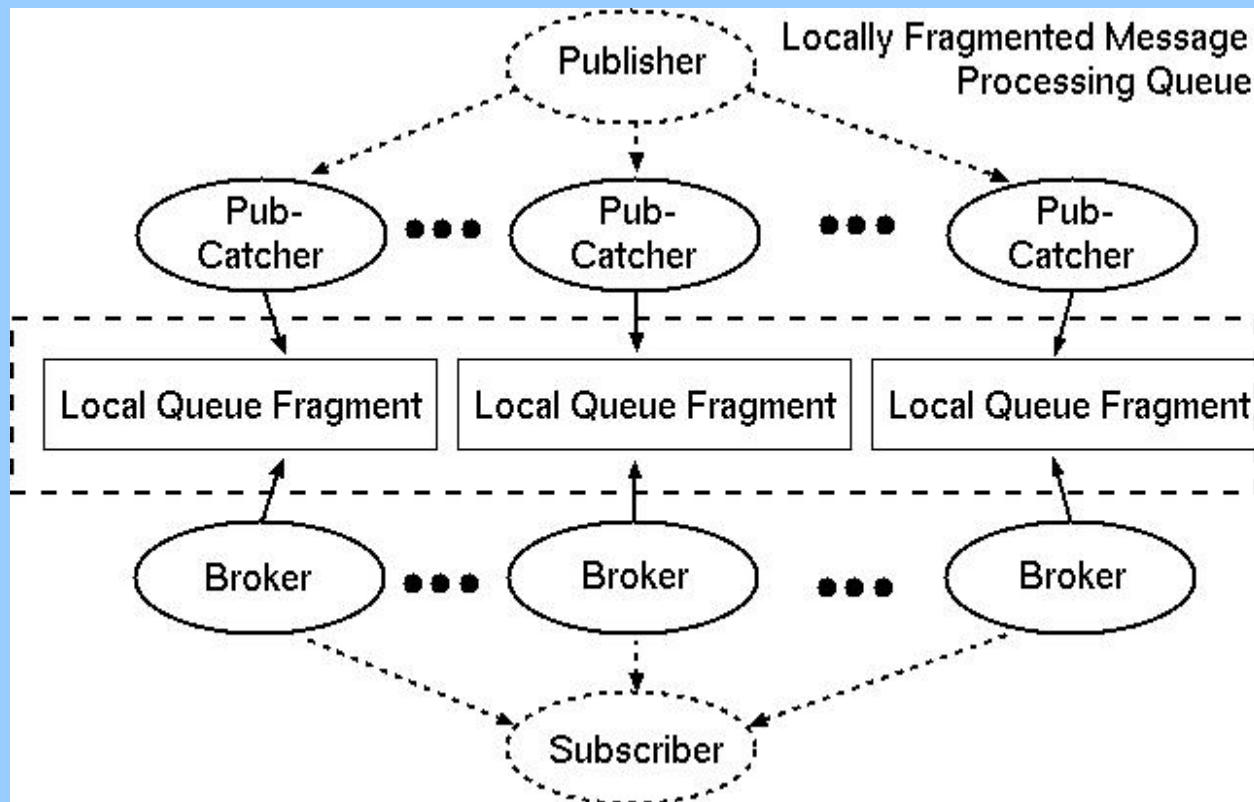
- Language Comparison

- Conclusion

Pub/Sub Model - GPAS

Pub/Sub Model - Fragmented

Chapel, X10, MPI

Chapel	X10	MPI
Used Chapel compiler version 0.6.	x10-1.0.47 used in this study.	MPICH 2.0 verson 1.0.6. Current is

Chapel 0.7   Current is   now 1.0.7

Documentation/Help

Chapel	X10	MPI
Ongoing email group. Some	Newsgroups. A lot of examples.	Mature Documentati on

examples.

Data Distribution

Chapel	X10	MPI
Define data distributions separate	Define data distributions separate	Collections are built from local

from variable   from variable   component

Object-Oriented

Chapel	X10	MPI
Inheriting from multiple base classes,	Interfaces (Java approach to	Used Mpich2.0 not OO (OOMPI
with	multiple	or MPI C++

Synchronization

Chapel	X10	MPI
"Sync" variables and "atomic" statement	"async" and "finish" operations on remote data	"barriers" and "communicat ors" for

Running the Code

Chapel	X10	MPI
Type the compiled executable	Use "testScript", find output in	Use Mpich2 scripts to start mpd
name.	log and error	and mpiexec

Sharing Data

Chapel	X10	MPI
Transparent	Transparent	Workers
Global	Global	share data
Partitioned	Partitioned	by "send",

Address   Address   "receive"

Chapel Language Experience

- Offers a variety of mechanisms that can assure cooperating processes are synchronized.

- *Sync* variables were used to synchronize producers and consumers. *Sync* and *single* variables are easy to use and an interesting feature of the Chapel language.

- Could not use Chapel "atomic" sections because they are not yet implemented.

- Time related functions, like *sleep*, are supported through a Time module. External function call mechanism also supported.

- Added synchronization for multiple brokers.

- Easy to extend synchronization for more complex pub/sub model.

- Easy to extend pub/sub example code to experiment with other built-in data distributions.

X10 Language Experience

- This study was unable to get Eclipse set up yet to work with the X10 code.

- X10 has a lot of built-in functionality, including most of Java. X10 is implemented as an extension of the Java language, replacing some of its functionality. Also inherited disadvantages of Java for high performance computing, including garbage collection and lack of pointers.

- We worked with array distribution code examples and then built a process distribution version. X10 has a lot of examples to work with. The runtime environment, based on their test harness was easy to work with, but put all output into a test results file. This is reasonable for checking to be sure that all test codes work correctly for each build.

- We implemented code to print the processing locations where operations take place and verified that operations were taking place at each processor at the correct time.